Computing the *k*-th Permutation of *n* Elements: A Factorial Number System Approach

Behrad Mahboobi
Electrical engineering department
Science and research branch of Azad university
b.mahboobi@gmail.com

***Abstract*—Determining the *k*-th permutation of *n* distinct elements in lexicographic order is a classic problem in combinatorics with a wide range of practical applications. This article explores a mathematically grounded approach using the factorial number system, a special case of mixed radix representation. We present a complete algorithm and demonstrate its correctness with a step-by-step example. While the method runs in $O(n^2)$, it is intuitive and practically useful. We also announce an upcoming article that will introduce an optimized linear-time $O(n)$ algorithm. This work is intended to provide theoretical insight and computational tools for practitioners and researchers alike.**

I. Introduction

Given n distinct elements, our goal is to compute the k-th permutation in lexicographic order, where $k \in [0, n! - 1]$. Rather than generating all $n!$ permutations, we can compute the desired one directly using the *factorial number system*, also known as the *factoradic*—a type of mixed radix numeral system.

This approach translates the problem of indexing permutations into a numeric representation problem, offering both elegance and computational efficiency. Though this method operates in $O(n^2)$ time, it lays the foundation for a faster $O(n)$ algorithm, which we will introduce in a subsequent article.

II. Factoradic Representation

The factorial number system represents a non-negative integer k using factorials instead of powers of a fixed base. Specifically, any integer $0 \leq k < n!$ can be uniquely expressed as:

$$k = a_{n-1}(n-1)! + a_{n-2}(n-2)! + \cdots + a_1(1)! + a_0(0)! \quad (1)$$

subject to:

$$0 \leq a_i \leq i \quad \text{for } i = 0, 1, \ldots, n-1 \quad (2)$$

To compute these coefficients, define $b_n = k$, then iteratively compute:

$$a_{n-j} = \left\lfloor \frac{b_{n-j+1}}{(n-j)!} \right\rfloor \quad (3)$$

$$b_{n-j} = b_{n-j+1} \bmod (n-j)! \quad (4)$$

III. Building the Permutation

Once the coefficients a_i are known, we can construct the permutation using the following iterative selection process:

- Initialize a list of available elements: $\Pi_0 = [0, 1, \ldots, n-1]$
- For each $k = 0$ to $n - 1$:

$$\pi_k = \Pi_k[a_k] \quad (5)$$

$$\Pi_{k+1} = \Pi_k \setminus \{\pi_k\} \quad (6)$$

After n steps, the sequence $[\pi_0, \pi_1, \ldots, \pi_{n-1}]$ is the desired permutation.

IV. Worked Example: 14-th Permutation of 4 Elements

Let $n = 4$, $k = 14$. We compute the factoradic representation:

$$a_3 = \left\lfloor \frac{14}{3!} \right\rfloor = 2, \qquad b_3 = 14 \bmod 6 = 2$$

$$a_2 = \left\lfloor \frac{2}{2!} \right\rfloor = 1, \qquad b_2 = 2 \bmod 2 = 0$$

$$a_1 = \left\lfloor \frac{0}{1!} \right\rfloor = 0, \qquad b_1 = 0 \bmod 1 = 0$$

$$a_0 = \left\lfloor \frac{0}{0!} \right\rfloor = 0$$

Factoradic digits: $(2, 1, 0, 0)$

Now construct the permutation step-by-step:

- $\Pi_0 = [0, 1, 2, 3]$, $a_3 = 2 \Rightarrow \pi_0 = 2$, $\Pi_1 = [0, 1, 3]$
- $a_2 = 1 \Rightarrow \pi_1 = 1$, $\Pi_2 = [0, 3]$
- $a_1 = 0 \Rightarrow \pi_2 = 0$, $\Pi_3 = [3]$
- $a_0 = 0 \Rightarrow \pi_3 = 3$

Thus, the 14-th permutation is: $[2, 1, 0, 3]$

V. Applications

This method has broad applications across computer science and mathematics:

- **Combinatorial Enumeration**: Efficiently map between indices and permutations.
- **Algorithm Design**: Useful in exhaustive search, test case generation, and ranking/unranking algorithms.
- **Cryptography**: Permutations serve as keys or transformation sequences.

- **Parallel Scheduling**: Permutations model execution orders in task scheduling.
- **Genetic Algorithms**: Permutations as chromosomes in combinatorial optimization.
- **Compact Representations**: Encode permutations using a single integer.

VI. Conclusion and Outlook

We presented a complete and intuitive algorithm to compute the *k*-th permutation of *n* elements using a factorial number system—a special case of mixed radix representation. While the time complexity is $O(n^2)$, the clarity and generality of the method make it highly valuable.

In our upcoming article, we will introduce a linear-time $O(n)$ version of this algorithm using more sophisticated data structures to eliminate the list update bottleneck. This promises to make the method scalable for much larger values of *n*.